Fleet Optimizer - Documento de Arquitectura

Introducción

En este documento describimos la arquitectura de Fleet Optimizer (FO). FO consume los datos de las cargas de un camion dentro de una ruta y en base a estos estima el tiempo promedio de cada viaje en esa ruta. Con los tiempos estimados por ruta, la lista de destinos y la lista de camiones, FO calcula una asignación de un numero de camiones a cada segmento de tiempo considerando las siguientes restricciones:

- Minimizar los viajes de los camiones
- Demanda de tonelaje
- Continuidad de asignación
- Conservación de la flota
- Tamaño de la flota

Los detalles formales del problema y el modelado pueden ser encontrados en https://www.researchgate.net/project/Collective-Intelligence-for-Fleet-Optimization-in-Mines

El resto del documento describe a alto nivel los componentes del sistema, sus interacciones y sus requerimientos.

Fleet Optimizer formulado como un proceso de búsqueda

Una alternativa para optimizar la asignación de la flota junto como un proceso de exploración de todas las alternativas que podrían generarse en cada paso del proceso.

El producto de este proceso es una simulación que resulta en una solución optima con respecto al numero de viajes realizados para cubrir la demanda de tonelaje asignada a cada ruta en la mina.

El proceso es modelado abstractamente con la jerarquía de clases que corresponde uno a uno con sus elementos: Los destinos de la mina (garaje, palas, trituradoras), las rutas que conectan dos elementos, los carros que transitan las rutas, etc. Cada uno de estos elementos tiene sus propiedades, por ejemplo: Los carros tienen capacidad en toneladas y los destinos tienen capacidad para flota residente.

El proceso se desenvuelve en tiempo discreto. Cada paso representa un segmento de tiempo en el mismo sentido que la formulación del problema como un programa lineal.

Dados los elementos, definimos el estado del problema como: *La ubicación de los carros en los destinos de la mina, el segmento de tiempo actual y la demanda de tonelaje satisfecha hasta el momento.*

En cada segmento de tiempo tenemos una serie de acciones a considerar. Las acciones incurren en un *costo* y resultan en un cambio del estado del problema. Concretamente, las acciones corresponden a cualquier asignación posible de los carros a las rutas de la mina.

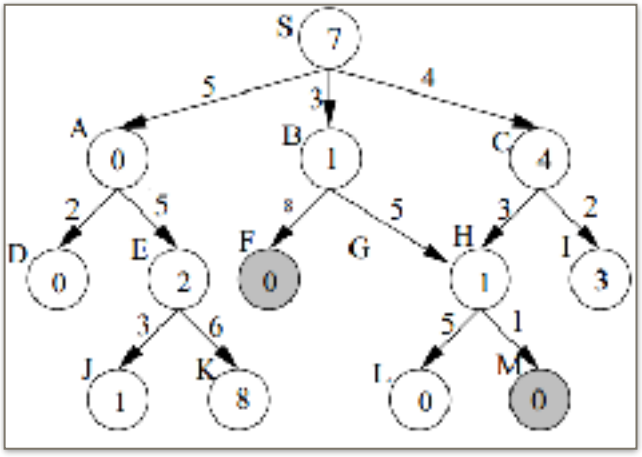
Tomando como ejemplo la mina del documento de Alfonso Bonillas, en el primero paso de la simulación, tenemos siete destinos y veintiún carros, todos residentes en el garaje. Las acciones posibles en el paso inicial son cualquier combinación de carros asignados a cualquiera de los destinos. El total de acciones posibles, si no existiera ninguna restricción, es $\binom{21}{4}$ y el resultado de cada una es una nueva distribución de los carros en los destinos de la mina. Esto podría resultar en una explosion combinatorial, sin embargo, restringimos la cantidad de camiones asignables a cada ruta con la capacidad de residencia en cada destino.

En el segundo paso, los carros comienzan a circular su ruta durante el tiempo o a trasladarse a otro destino. Si los carros transportan carga, el resultado de la acción satisface parcialmente la demanda de esa ruta en particular y el carro permanece en la misma ubicación donde comenzó este segmento de tiempo, pero si se resigna a otra ruta, no decremento la demanda de la ruta en la que pertenecía, pero termina el segmento de tiempo en una nueva ubicación.

Cada una de estas acciones incurre en un costo, que es un viaje, y esta es la cantidad que deseamos minimizar eligiendo las acciones que satisfagan la demanda de todas las rutas utilizando los carros adecuados.

Al final del proceso, en el ultimo segmento de tiempo, todos los carros deben de volver al garaje, para cerrar el turno.

Visualizando el proceso como un árbol de decision, consideremos que cada nodo corresponde a un posible estado del problema al iniciar el turno. En el primer segmento de tiempo solo hay un estado, el inicial, el cual es la raíz del árbol de decision. Cada posible acción representa una rama del nodo actual cuyo desenlace es el estado resultante.



Conforme el proceso se desenvuelva con respecto a su dinámica, eventualmente el árbol de decision llegara a un final. Todos los posibles finales son las *hojas* del árbol y en el caso particular de *Fleet Optimizer*, estas hojas representan estados donde la demanda de todas las rutas ha sido satisfecha o se han agotado los segmentos de tiempo del turno.

No todos los estados finales son aceptables, solo aquellos que hayan satisfecho la demanda en todas las rutas y regresen todos los carros al garage (fleet conservation constraint).

Por cada hoja que represente una solución aceptable del problema, los pasos para llegar a ella son la secuencia de acciones tomadas desde el inicio de la simulación hasta llegar a esa hoja del árbol en particular. Esta secuencia de acciones tiene un *costo.* El costo es el numero de viajes realizados y las mejores soluciones son aquellas cuyo costo total, el costo acumulado de cada acción, sea menor.

Búsqueda de la solución.

Cualquier algoritmo que busque una mejor solución tiene que recorrer el árbol de decisiones del proceso hasta encontrar una mejor solución. Una exhaustiva no es factible dado a la explosion exponencial de las posibilidades a considerar por cada nivel del árbol.

Una búsqueda mas eficiente da prioridad los elementos resultantes de las acciones de menor costo, en este caso las acciones que minimizan el numero de viaje, garantizando que en el momento en que se encuentre una solución, esta sera una solución optima.

Uniform Cost Search (UCS) es un algoritmo eficiente para considerar las secuencias de acciones siguiendo este criterio utilizando una cola de prioridad utilizando como criterio el costo de cada camino en el árbol.

*A** es una extension de *UCS,* que ademas del costo utiliza una *heurística* para estimar el costo restante para alcanzar la solución optima. Esta heurística debe de ser expresada en términos compatibles con el costo (peras con peras, manzanas con manzanas) y no debe de sobre estimar el costo optimo para no romper con la garantía de optimalidad. Si puede relajar el requerimiento de optimalidad, es aceptable sobreestimar el costo restante, siempre y cuando la heurística sea informativa del costo restante real.

Código: Core_Search

Core_Search es un paquete python que toma los datos de la flota y los parámetros del problema y ejecuta una simulación del proceso de la mina.

La mina esta modelada como una red, que conecta elementos de la mina como palas y trituradoras por medio de *rutas*. Es indispensable que la mina tenga un elemento llamado *grage,* el cual es punto de reunion para la flota de camiones durante el inicio y final del turno. Cada ruta puede tener una demanda de tonelaje.

La mina también cuenta con una flota de camiones, el cual cada uno tiene una capacidad particular en toneladas

Core_Seach toma la *configuración* de la mina como una lista de conexiones, que son representadas como **tuplas** de python, cuyos elementos son las maquinas en la mina que estan conectadas entre si. Por ejemplo:

```
# First build the locations
shovel1 = Location("S1", 2)
shovel2 = Location("S2", 2)
loader1 = Location("L1", 2)
loader2 = Location("L2", 2)
waste_dump = Location("W", 2)
crusher = Location("C", 2)
garage = Location("garage", 21)

# Build the mine configuration
connections = [
    (garage, loader1),
    (garage, loader2),
    (loader1, garage),
    (loader2, garage),
    (waste_dump, garage),
    (garage, shovel1),
    (garage, shovel2),
    (waste_dump, loader1),
    (loader1, waste_dump),
    (waste_dump, shovel1),
    (shovel1, waste_dump),
    (waste_dump, shovel2),
    (shovel2, waste_dump),
    (crusher, shovel1),
    (shovel1, crusher),
    (crusher, shovel2),
    (shovel2, crusher),
    (crusher, loader2),
    (loader2, crusher),
    (loader1, crusher)
```

La flota de camiones esta definida como una secuencia de objetos de tipo **Truck**:

```
# Generate the trucks
trucks = [Truck("truck_%i" % i, c) for i, c in zip(range(1, num_trucks+1), it.cycle([100]))]
```

La demanda en tonelaje de cada ruta esta definida como un diccionario, donde la llave es la ruta y el valor es la demanda en toneladas:

```
demands = OrderedDict(
[((shovel1, crusher), 8000),
    ((shovel2, crusher), 1200),
    ((loader1, crusher), 4000),
    ((shovel1, waste_dump), 1600),
    ((shovel2, waste_dump), 2000),
    ((loader1, waste_dump), 1000)]
)
```

Por ultimo, los elementos de la simulación se integran dentro de un objeto que representa el *estado* de la simulación:

```
# Create the initial state
initial_state = FleetState(config, trucks, demands, num_segments)
```

El estado inicial representa el estado de la mina al principio de un turno, con todos los camiones en el garage y todas las demandas de las rutas pendientes por cubrir. Para iniciar la simulación, *Core_Search* cuenta con una implementación del algoritmo **A*** que ejecuta la simulación, buscando el estado que cubra todas las demandas de las rutas, minimizando el numero de viajes necesarios:

```
searcher = AStar(initial_state, heuristic, listener)
solution = searcher.solve()
```

Si la simulación encuentra una solución, regresa el estado final de la simulación, del cual se puede reconstruir la secuencia de acciones utilizadas para alcanzarlo. Si la simulación no encuentra una solucion, regresa *None.*

El ejemplo detallado de como correr una simulación se puede encontrar en el archivo *core_search/run.py*

Código: LeanUI

El modulo *lean_ui* es un portal hecho con **Django** que imita al script *run.py*, pero es un ejemplo de como integrar *core_search* a dentro de un sistema de informacion. Muestra exactamente la misma información, pero dentro de una pagina web rudimentaria encontrada en: http://127.0.0.1:8000/fleet/

Para que el proyecto corra, el modulo *core_search* tiene que estar accesible desde **PYTHONPATH.** Las dependencias de este modulo están en el archivo “requirements.txt” dentro de *lean_ui*